Intersection of Convex Objects: The Method of Separating Axes

David Eberly
Geometric Tools, LLC

Created: January 28, 2001
Last Modified: March 2, 2008

Contents

1 Introduction

This document describes the *method of separating axes*, a method for determining whether or not two stationary convex objects are intersecting. The ideas can be extended to handle moving convex objects and are useful for predicting collisions of the objects and for computing the first time of contact. The current focus of this document is on the *test intersection* geometric query, a query that just indicates whether or not an intersection exists or will occur when the objects are moving. The problem of computing the set of intersection is denoted a *find intersections* geometric query and is generally more difficult to implement than the *test intersection* query. Information from the *test* query can help determine the contact set that the *find* query must construct. This document will describe in what way the *test* query information can be used.

2 Separation by Projection onto a Line

A test for nonintersection of two convex objects is simply stated: If there exists a line for which the intervals of projection of the two objects onto that line do not intersect, then the objects are do not intersect. Such a line is called a *separating line* or, more commonly, a *separating axis*. The translation of a separating line is also a separating line, so it is sufficient to consider lines that contain the origin. Given a line containing the origin and with unit-length direction $\mathbf{D}$, the projection of a compact[1], convex[2] set C onto the line is the interval

$$I = [\lambda_{\min}(\mathbf{D}), \lambda_{\max}(\mathbf{D})] = [\min\{\mathbf{D} \cdot \mathbf{X} : \mathbf{X} \in C\}, \max\{\mathbf{D} \cdot \mathbf{X} : \mathbf{X} \in C\}].$$

Two compact convex sets C_0 and C_1 are separated if there exists a direction $\mathbf{D}$ such that the projection intervals I_0 and I_1 do not intersect. Specifically they do not intersect when

$$\lambda_{\min}^{(0)}(\mathbf{D}) > \lambda_{\max}^{(1)}(\mathbf{D}) \text{ or } \lambda_{\max}^{(0)}(\mathbf{D}) < \lambda_{\min}^{(1)}(\mathbf{D}). \tag{1}$$

The superscript corresponds to the index of the convex set. Although the comparisons are made where $\mathbf{D}$ is unit length, the comparison results are invariant to changes in length of the vector. This follows from $\lambda_{\min}(t\mathbf{D}) = t\lambda_{\min}(\mathbf{D})$ and $\lambda_{\max}(t\mathbf{D}) = t\lambda_{\max}(\mathbf{D})$ for $t > 0$. The Boolean value of the pair of comparisons is also invariant when $\mathbf{D}$ is replaced by the opposite direction $-\mathbf{D}$. This follows from $\lambda_{\min}(-\mathbf{D}) = -\lambda_{\max}(\mathbf{D})$ and $\lambda_{\max}(-\mathbf{D}) = -\lambda_{\min}(\mathbf{D})$. When $\mathbf{D}$ is not unit length, the intervals obtained for the separating axis tests are not the projections of the object onto the line, rather they are scaled versions of the projection intervals. I make no distinction in this document between the scaled projection and regular projection. I will also use the terminology that the direction vector for a separating axis is called a *separating direction*, a direction that is not necessarily unit length.

# 3 Separation of Convex Polygons in 2D

For a pair of convex polygons in 2D, only a finite set of direction vectors needs to be considered for separation tests. That set includes the normal vectors to the edges of the polygons. The left picture in Figure 1 shows

---

[1] A set is *compact* if it is closed and bounded. To illustrate in one dimension, the interval $[0, 1]$ is closed and bounded, so it is compact. The interval $[0, 1)$ is bounded, but not closed since it does not contain the limiting point 1, so it is not compact. The interval $[0, \infty)$ is closed, but not bounded, so it is not compact.

[2] A set is *convex* if given any two points $\mathbf{P}$ and $\mathbf{Q}$ in the set, the line segment $(1 - t)\mathbf{P} + t\mathbf{Q}$ for $t \in [0, 1]$ is also in the set. In one dimension, both $[0, 1]$ and $[0, 1)$ are convex.

two nonintersecting polygons that are separated along a direction determined by the normal to an edge of one polygon. The right picture shows two polygons that intersect (there are no separating directions).

Figure 1 Nonintersecting convex polygons (left). Intersecting convex polygons (right).
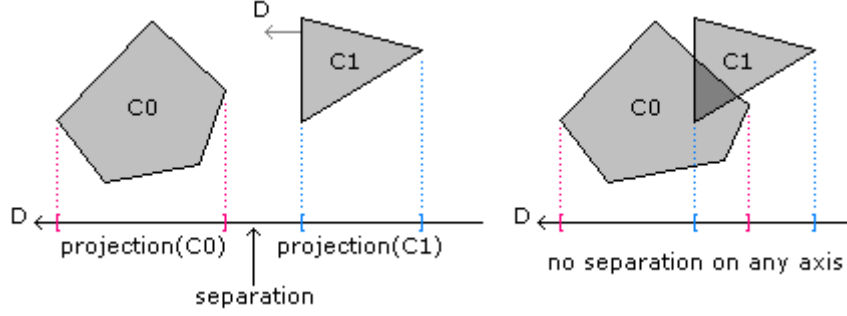
The intuition for why only edge normals must be tested is based on having two convex polygons just touching with no interpenetration. Figure 2 shows the three possible configurations: edge-edge contact, vertex-edge contact, and vertex-vertex contact.

Figure 2 Edge-edge contact (left), vertex-edge contact (middle), and vertex-vertex contact.

The blue lines are perpendicular to the separation lines that would occur for the object translated away from the other by an infinitesimal distance.

Let C_j, $j = 0$ or $j = 1$, be the convex polygons with vertices $\{\mathbf{V}_i^{(j)}\}_{i=0}^{N_j-1}$ that are counterclockwise ordered. The direct implementation for a separation test for direction $\mathbf{D}$ just computes the extreme values of the projection and compares them. That is, compute $\lambda_{\min}^{(j)}(\mathbf{D}) = \min_{0 \le i < N_0}\{\mathbf{D} \cdot \mathbf{V}_i^{(j)}\}$ and $\lambda_{\max}^{(j)}(\mathbf{D}) = \max_{0 \le i < N_1}\{\mathbf{D} \cdot \mathbf{V}_i^{(j)}\}$ and test the inequalities in Equation 1. However, the pseudocode shown below avoids projecting all the vertices for both polygons by only testing if the polygons are on opposite sides of the test line. This is a reasonable modification when the polygons have a large number of nonparallel edges. For triangles or rectangles, the direct implementation is a better choice. In particular it is a better choice when the rectangles are represented by a center point, two orthonormal axes, and two half-width measurements; the projection intervals are trivial to compute with this representation. Given an edge (x, y), an outward pointing normal is $(x, y)^\perp = (y, -x)$. In the pseudocode, this operation is denoted by `Perp`.

```
int WhichSide (PointSet S, Point D, Point P)
```

3

```
{
    // S vertices are projected to the form P+t*D.  Return value is +1 if all t > 0,
    // -1 if all t < 0, 0 otherwise, in which case the line splits the polygon.

    positive = 0;  negative = 0;
    for (i = 0; i < C.N; i++)
    {
        t = Dot(D,S.V(i)-P);
        if ( t > 0 ) positive++; else if ( t < 0 ) negative++;
        if ( positive && negative ) return 0;
    }
    return ( positive ? +1 : -1 );
}

bool TestIntersection2D (ConvexPolygon C0, ConvexPolygon C1)
{
    // Test edges of C0 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C0 is [m,0] where m <= 0.  Only try to determine
    // if C1 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C0.N-1; i0 < C0.N; i1 = i0, i0++)
    {
        D = Perp(C0.V(i0) - C0.V(i1));
        if ( WhichSide(C1.V,D,C0.V(i0)) > 0 )
        {   // C1 is entirely on 'positive' side of line C0.V(i0)+t*D
            return false;
        }
    }

    // Test edges of C1 for separation.  Because of the counterclockwise ordering,
    // the projection interval for C1 is [m,0] where m <= 0.  Only try to determine
    // if C0 is on the 'positive' side of the line.
    for (i0 = 0, i1 = C1.N-1; i0 < C1.N; i1 = i0, i0++)
    {
        D = Perp(C1.V(i0) - C1.V(i1));
        if ( WhichSide(C0.V,D,C1.V(i0)) > 0 )
        {   // C0 is entirely on 'positive' side of line C1.V(i0)+t*D
            return false;
        }
    }

    return true;
}
```

4 Separation of Convex Polyhedra in 3D

For a pair of convex polyhedra in 3D, only a finite set of direction vectors needs to be considered for separation tests. That set includes the normal vectors to the faces of the polyhedra and vectors generated by a cross product of two edges, one from each polyhedron. The intuition is similar to that of convex polygons in 2D. If the two polyhedra are just touching with no interpenetration, then the contact is one of face-face, face-edge, face-vertex, edge-edge, edge-vertex, or vertex-vertex.

Let C_j, $j = 0$ or $j = 1$, be the convex polyhedra with vertices $\{\mathbf{V}_i^{(j)}\}_{i=0}^{N_j-1}$, edges $\{\mathbf{E}_i^{(j)}\}_{i=0}^{M_j-1}$, and faces $\{\mathbf{F}_i^{(j)}\}_{i=0}^{L_j-1}$. Let the faces be planar convex polygons whose vertices are counterclockwise ordered as you view the face from outside the polyhedron. Outward pointing normal vectors can be stored with each face as a way of storing the orientation. The pseudocode for 3D that is similar to that in 2D is shown below. It is assumed that: (1) each face has queries that allow access to the face normal and to a vertex on the face and (2) each edge has a query that allows access to a vertex on the edge.

```
bool TestIntersection3D (ConvexPolyhedron C0, ConvexPolyhedron C1)
{
    // Test faces of C0 for separation.  Because of the counterclockwise ordering,
```

```
// the projection interval for C0 is [m,0] where m <= 0.  Only try to determine
// if C1 is on the 'positive' side of the line.
for ( i = 0;  i < C0.L;  i++)
{
    D = C0.F( i ).normal;   // outward pointing
    if ( WhichSide(C1.V,D,C0.F( i ).vertex) > 0 )
    {   // C1 is entirely on 'positive' side of line C0.F( i ).vertex+t*D
        return false ;
    }
}

// Test faces of C1 for separation.  Because of the counterclockwise ordering ,
// the projection interval for C1 is [m,0] where m <= 0.  Only try to determine
// if C0 is on the 'positive' side of the line.
for ( i = 0;  i < C1.L;  i++)
{
    D = C1.F( i ).normal;   // outward pointing
    if ( WhichSide(C0.V,D,C1.F( i ).vertex) > 0 )
    {   // C0 is entirely on 'positive' side of line C1.F( i ).vertex+t*D
        return false ;
    }
}

// Test cross product of pairs of edges, one from each polyhedron.
for ( i = 0;  i < C0.M;  i++)
{
    for ( j = 0;  j < C1.M;  j++)
    {
        D = Cross (C0.E( i ),C1.E( j ));

        int side0 = WhichSide(C0.V,D,C0.E( i ).vertex );
        if ( side0 == 0)
        {
            continue ;
        }

        int side1 = WhichSide(C1.V,D,C0.E( i ).vertex );
        if ( side1 == 0 )
        {
            continue ;
        }

        if ( side0*side1 < 0 )
        {   // C0 and C1 are on 'opposite' sides of line C0.E( i ).vertex+t*D
            return false ;
        }
    }
}

return true ;
}
```

The `WhichSide` function is the same as the one for two dimensions, except that the input is now three dimensional.

5 Separation of Convex Polygons in 3D

For a pair of convex polygons in 3D, again only a finite set of direction vectors needs to be considered for separation tests. Let C_j, $j = 0$ or $j = 1$ be the convex polygons with vertices $\{\mathbf{V}_i^{(j)}\}_{i=0}^{M_j-1}$ with the convention that $\mathbf{V}_{M_j}^{(j)} = \mathbf{V}_0^{(j)}$. Each set of vertices is necessarily coplanar. Let $\mathbf{N}^{(j)}$ be normal vectors for those planes. The edges for the polygons are $\mathbf{E}_i^{(j)} = \mathbf{V}_{i+1}^{(j)} - \mathbf{V}_i^{(j)}$ of $0 \leq i < M_j$.

The polygons normals are potential separating directions. If either normal direction separates the polygons,

then no intersection occurs. However, if neither normal direction separates the polygons, then two possibilities exist for the remaining potential separating directions. The first case is that the polygons are coplanar (the 2D case) and the remaining potential separating directions are those vectors in the common plane and perpendicular to the triangle edges, $\mathbf{N}^{(j)} \times \mathbf{E}_i^{(j)}$ for $0 \leq j \leq 1$ and $0 \leq i < M_j$, a total of $2M_j$ vectors. The second case is that the polygon planes are not parallel and that each polygon is split by the plane of the other polygon. The remaining potential separating directions are the cross products $\mathbf{E}_i^{(0)} \times \mathbf{E}_j^{(1)}$ for $0 \leq i < M_0$ and $0 \leq j < M_1$, a total of $M_0 M_1$ vectors.

The pseudocode for intersection testing is

```
bool TestIntersection3D (ConvexPolygon C0, ConvexPolygon C1)
{
    // test normal for C0
    if ( WhichSide(C1,C0.N,C0.V(0)) != 0 )
    {   // C1 is entirely on one side of plane of C0
        return false;
    }
}
```

6 Separation of Moving Convex Objects

The method of separating axes can be extended to handle convex objects moving with constant velocity. If C_0 and C_1 are convex objects with velocities $\mathbf{W}_0$ and $\mathbf{W}_1$, then it can be determined via projections if the objects will intersect for some time $T \geq 0$. Moreover, if they do, the first time of contact can be computed. Without loss of generality, it is enough to work with a stationary object C_0 and a moving object C_1 with velocity $\mathbf{W}$ since one can always use $\mathbf{W} = \mathbf{W}_1 - \mathbf{W}_0$ to perform the calculations as if C_0 were not moving.

If the C_0 and C_1 are initially intersecting, then the first time of contact is $T = 0$. Otherwise the convex objects are initially disjoint. The projection of C_1 onto a line with direction $\mathbf{D}$ not perpendicular to $\mathbf{W}$ is itself moving. The speed of the projection along the line is $\omega = (\mathbf{W} \cdot \mathbf{D})/|\mathbf{D}|^2$. If the projection interval of C_1 moves away from the projection interval of C_0, then the two objects will never intersect. The interesting cases are when the projection intervals for C_1 move towards those of C_0.

The intuition for how to predict an intersection is much like that for selecting the potential separating directions in the first place. If the two convex objects intersect at a first time $T_{\text{first}} > 0$, then their projections are not separated along any line. An instant before first contact, the objects are separated. Consequently there must be at least one separating direction for the objects for $T_{\text{first}} - \varepsilon$ for small $\varepsilon > 0$. Similarly, if the two convex objects intersect at a last time $T_{\text{last}} > 0$, then their projections are also not separated at that time along any line, but an instant after last contact, the objects are separated. Consequently there must be at least one separating direction for the objects for $T_{\text{last}} + \varepsilon$ for small $\varepsilon > 0$. Both T_{first} and T_{last} can be tracked as each potential separating axis is processed. After all directions are processed, if $T_{\text{first}} \leq T_{\text{last}}$, then the two objects do intersect with first contact time T_{first}. It is also possible that $T_{\text{first}} > T_{\text{last}}$ in which case the two objects cannot intersect.

Let S_0 and S_1 denote the set of potential separating directions and let $\mathbf{W}_0$ and W_1 denote the velocities for C_0 and C_1, respectively. Pseudocode for testing for intersection of two moving convex objects is given below. The time interval over which the event is of interest is $[0, T_{\max}]$. If knowing an intersection at *any* future time is desired, then set $T_{\max} = \infty$. Otherwise, $T_{\max}$ is finite. The function possibly can indicate there is no intersection on $[0, T_{\max}]$, even though there could be an intersection at some time $T > T_{\max}$.

```
bool ObjectsIntersect (Convex C0, Convex C1, float TMax, float& TFirst, float& TLast)
```

```
{
    W = C1.W − C0.W;   // process as if C0 is stationary, C1 is moving
    S = Union(C0.S,C1.S);   // all potential separating axes
    TFirst = 0;   TLast = INFINITY;
    for each D in S do
    {
        speed = Dot(D,W);
        min0 = minimum(Dot(D,C0.V(i)));   max0 = maximum(Dot(D,C0.V(i)));
        min1 = minimum(Dot(D,C1.V(i)));   max1 = maximum(Dot(D,C1.V(i)));
        if ( max1 < min0 )   // interval(C1) initially on 'left' of interval(C0)
        {
            if ( speed <= 0 ) return false;   // intervals moving apart
            T = (min0 − max1)/speed;   if ( T > TFirst ) TFirst = T;
            if ( TFirst > TMax ) return false;   // 'quick out'
            T = (max0 − min1)/speed;   if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;   // 'quick out'
        }
        else if ( max0 < min1 )   // interval(C1) initially on 'right' of interval(C0)
        {
            if ( speed >= 0 ) return false;   // intervals moving apart
            T = (max0 − min1)/speed;   if ( T > TFirst ) TFirst = T;
            if ( TFirst > TMax ) return false;   // 'quick out'
            T = (min0 − max1)/speed;   if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;   // 'quick out'
        }
        else   // interval(C0) and interval(C1) overlap
        {
            if ( speed > 0 )
            {
                T = (max0 − min1)/speed;   if ( T < TLast ) TLast = T;
                if ( TFirst > TLast ) return false;   // 'quick out'
            }
            else if ( speed < 0 )
            {
                T = (min0 − max1)/speed;   if ( T < TLast ) TLast = T;
                if ( TFirst > TLast ) return false;   // 'quick out'
            }
        }
    }
    return true;
}
```

The following example illustrates the ideas. The first box is the unit cube $0 \leq x \leq 1$ and $0 \leq y \leq 1$ and is stationary. The second box is initially $0 \leq x \leq 1$ and $1 + \delta \leq y \leq 2 + \delta$ for some $\delta > 0$. Let its velocity be $(1, -1)$. Whether or not the second box intersects the first box depends on the value of δ. The only potential separating axes are $(1, 0)$ and $(0, 1)$. Figure 3 shows the initial configuration for three values of δ, one where there will be an edge-edge intersection, one where there will be a vertex-vertex intersection, and one where there is no intersection.

Figure 3 Edge-edge intersection predicted (left). Vertex-vertex intersection predicted (middle). No intersection predicted (right).
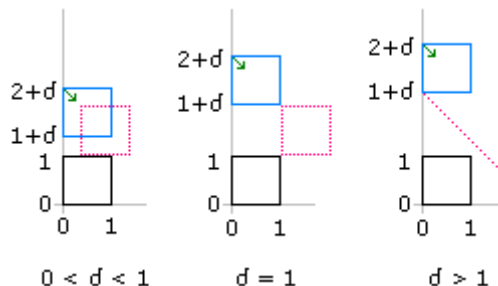
The black box is stationary. The blue box is moving. The green vector indicates the direction of motion. The red boxes indicate where the moving box first touches the stationary box. In the right image in the figure, the dotted red line indicates that the moving box will miss the stationary box. For $\mathbf{D} = (1, 0)$, the pseudocode produces `min0 = 0`, `max0 = 1`, `min1 = 0`, `max1 = 1`, and `speed = 1`. The projected intervals are initially overlapping. Since the speed is positive, `T = (max0-min1)/speed = 1 < TLast = INFINITY` and `TLast` is updated to 1. For $\mathbf{D} = (0, 1)$, the pseudocode produces `min0 = 0`, `max0 = 1`, `min1 = 1+delta`, `max1 = 2+delta`, and `speed = -1`. The moving projected interval is initially on the right of the stationary projected interval. Since the speed is negative, `T = (max0-min1)/speed = delta > TFirst = 0` and `TFirst` is updated to `delta`. The next block of code sets `T = (min0-max1)/speed = 2+delta`. The value `TLast` is not updated since $2 + \delta < 1$ cannot happen for $\delta > 0$. On exit from the loop over potential separating directions, $T_{\text{First}} = \delta$ and $T_{\text{last}} = 1$. The objects intersect if and only if $T_{\text{first}} \leq T_{\text{last}}$, or $\delta \leq 1$. This condition is consistent with the images in Figure 3. The left image has $\delta < 1$ and the middle image has $\delta = 1$, intersections occurring in both cases. The right image has $\delta > 1$ and no intersection occurs.

7   Contact Set for Moving Convex Objects

Given two moving convex objects C_0 and C_1 with velocities $\mathbf{W}_0$ and $\mathbf{W}_1$ that are initially not intersecting, the material in the last section showed how to compute the first time of contact T, if it exists. Assuming it does, the sets $C_0 + T\mathbf{W}_0 = \{\mathbf{X} + T\mathbf{W}_0 : \mathbf{X} \in C_0\}$ and $C_1 + T\mathbf{W}_1 = \{\mathbf{X} + T\mathbf{W}_1 : \mathbf{X} \in C_1\}$ are just touching with no interpenetration. Figure 2 shows the various configurations for 2D.

The `ObjectsIntersect` function can be modified to keep track of which vertices, edges, and/or faces are projected to the end points of the projection interval. At the first time of contact, this information can be used to determine how the two objects are positioned with respect to each other. In 2D, if the contact is vertex-edge or vertex-vertex, then the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a line segment that contains at least one vertex. In 3D, if the contact is vertex-vertex, vertex-edge, or vertex-face, then the contact set is a single point, a vertex. If the contact is edge-edge, the contact set is a single point (the intersection of the two lines containing the edges) or a line segment (the edges are on the same line). If the contact is edge-face, then the contact set is a line segment. Otherwise the contact is face-face and the contact set is the intersection of two planar convex polygons.

8   Example: Two Moving Triangles in 2D

Consider two triangles, $\langle \mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2 \rangle$ and $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2 \rangle$, both having counterclockwise ordering. For the sake of indexing notation, define $\mathbf{U}_3 = \mathbf{U}_0$ and $\mathbf{V}_3 = \mathbf{V}_0$. The edges are $\mathbf{E}_i = \mathbf{U}_{i+1} - \mathbf{U}_i$ and $\mathbf{F}_i = \mathbf{V}_{i+1} - \mathbf{V}_i$ for $0 \leq i \leq 2$. Define $(x, y)^\perp = (y, -x)$. Outward pointing normals to the edges are $\mathbf{N}_i = \mathbf{E}_i^\perp$ and $\mathbf{M}_i = \mathbf{F}_i^\perp$ for $0 \leq i \leq 2$. The six normals are the potential separating directions. Let the triangle velocities be $\mathbf{W}_0$ and $\mathbf{W}_1$.

In the pseudocode for testing for intersection of two moving convex objects (in this case $C0$ and $C1$ represent the two triangles), the calculation of the minimum and maximum projections for the triangles can be done so that additional information is known about how the two triangles are oriented with respect to each other. Three cases occur for the projection:

1. Two vertices project to the minimum of the interval and one vertex projects to the maximum.

2. One vertex projects to the minimum of the interval and two vertices project to the maximum.

3. One vertex projects to the minimum, one vertex projects to the maximum, and one vertex projects to an interval interior point.

A flag can be associated with each triangle to indicate which of these three cases has occured for a given potential separating direction $\mathbf{D}$, call the flag values M21, M12, and M11. In addition the a flag and to the extreme values of the projection, it is necessary to remember the indices of the vertices that map to the extreme values. A convenient data structure is

```
ProjectionMap { M21, M12, M11 };
Configuration
{
    ProjectionMap map;
    int index[3];
    float min, max;
};
```

In the function `ObjectsIntersect` from the last section, two configuration objects are declared, `Cfg0` for the U triangle (the `C0` polygon) and `Cfg1` for the V triangle (the `C1` polygon). To illustrate for the specific case $\mathbf{D} = \mathbf{N}_0$, the block

```
min0 = minimum(Dot(D,U(i)));
max0 = maximum(Dot(D,U(i)));
min1 = minimum(Dot(D,V(i)));
max1 = maximum(Dot(D,V(i)));
```

where `U(i)` denotes $\mathbf{U}_i$ and `V(i)` denotes $\mathbf{V}_i$, is replaced by

```
// U2 maps to minimum, U0 and U1 map to maximum
Cfg0.map = M12;
Cfg0.index[0] = 2;   Cfg0.index[1] = 0;   Cfg0.index[2] = 1;
Cfg0.min = -Dot(N0,E2);
Cfg0.max = 0;

// Compute min and max of interval for second triangle.  Keep track of
// vertices that project to min and max.
d0 = Dot(N0,V0-U0);   d1 = Dot(N0,V1-U0);   d2 = Dot(N0,V2-U0);
if ( d0 <= d1 )
{
    if ( d1 <= d2 )   // d0 <= d1 <= d2
    {
        if ( d0 != d1 )
```

```
                Cfg1.map = ( d1 != d2 ? M11 : M12 );
            else
                Cfg1.map = M21;

            Cfg1.index[0] = 0;  Cfg1.index[1] = 1;  Cfg1.index[2] = 2;
            Cfg1.min = d0;  Cfg1.max = d2;
        }
        else if ( d0 <= d2 )  // d0 <= d2 < d1
        {
            if ( d0 != d2 )
            {
                Cfg1.map = M11;
                Cfg1.index[0] = 0;  Cfg1.index[1] = 2;  Cfg1.index[2] = 1;
            }
            else
            {
                Cfg1.map = M21;
                Cfg1.index[0] = 2;  Cfg1.index[1] = 0;  Cfg1.index[2] = 1;
            }

            Cfg1.min = d0;  Cfg1.max = d1;
        }
        else  // d2 < d0 <= d1
        {
            Cfg1.map = ( d0 != d1 ? M12 : M11 );
            Cfg1.index[0] = 2;  Cfg1.index[1] = 0;  Cfg1.index[2] = 1;
            Cfg1.min = d2;  Cfg1.max = d1;
        }
    }
}
else
{
    if ( d2 <= d1 )  // d2 <= d1 < d0
    {
        if ( d2 != d1 )
        {
            Cfg1.map = M11;
            Cfg1.index[0] = 2;  Cfg1.index[1] = 1;  Cfg1.index[2] = 0;
        }
        else
        {
            Cfg1.map = M21;
            Cfg1.index[0] = 1;  Cfg1.index[1] = 2;  Cfg1.index[2] = 0;
        }

        Cfg1.min = d2;  Cfg1.max = d0;
    }
    else if ( d2 <= d0 )  // d1 < d2 <= d0
    {
        Cfg1.map = ( fD2 != fD0 ? M11 : M12 );
        Cfg1.index[0] = 1;  Cfg1.index[1] = 2;  Cfg1.index[2] = 0;
        Cfg1.min = fD1;  Cfg1.max = fD0;
    }
    else  // d1 < d0 < d2
    {
        Cfg1.map = M11;
        Cfg1.index[0] = 1;  Cfg1.index[1] = 0;  Cfg1.index[2] = 2;
        Cfg1.min = fD1;  Cfg1.max = fD2;
    }
}
```

Similar blocks of code can be written for other potential separating directions.

The conditional statements comparing minima and maxima must be modified to keep track of the relative location of the moving interval to the stationary one. The input configurations are for the current potential separating axis. If this axis direction becomes the new candidate for first time of contact, we need to remember its configurations to be used in the function that computes the contact set. The pseudocode below stores the configurations, `TCfg0` and `TCfg1`, that correspond to the first time of contact as it is currently known.

```
// 'side' is initialized to NONE outside the loop over directions

if ( Cfg1.max < Cfg0.min )  // V-interval initially on 'left' of U-interval
{
    if ( speed <= 0 )  // intervals moving apart
        return false;

    // update first time of contact
    T = (Cfg0.min - Cfg1.max)/speed;
    if ( T > TFirst )
    {
        TFirst = T;
        side = LEFT;
        TCfg0 = Cfg0;
        TCfg1 = Cfg1;
    }

    // test for exceedance of time interval
    T = (Cfg0.max - Cfg1.min)/speed;
    if ( T < TLast )
        TLast = T;

    // test for separation
    if ( TFirst > TLast )
        return false;
}
else if ( max0 < min1 )  // V-interval initially on 'right' of U-interval
{
    if ( speed >= 0 )
        return false;  // intervals moving apart

    // update first time of contact
    T = (Cfg0.max - Cfg1.min)/speed;
    if ( T > TFirst )
    {
        TFirst = T;
        side = RIGHT;
        TCfg0 = Cfg0;
        TCfg1 = Cfg1;
    }

    // test for exceedance of time interval
    T = (Cfg0.min - Cfg1.max)/speed;
    if ( T < TLast )
        TLast = T;

    // test for separation
    if ( TFirst > TLast )
        return false;
}
else  // U-interval and V-interval overlap
{
    if ( speed > 0 )
    {
        // update last time
        T = (Cfg0.max - Cfg1.min)/speed;
        if ( T < TLast )
            TLast = T;

        // test for separation
        if ( TFirst > TLast )
            return false;
    }
    else if ( speed < 0 )
    {
        // update last time
        T = (Cfg0.min - Cfg1.max)/speed;
        if ( T < TLast )
            TLast = T;

        // test for separation
        if ( TFirst > TLast )
```

11

```
            return false ;
        }
    }
```

If $\mathbf{N}_0$ is the last separating axis an instant before first time of contact, then the `TConfig` string contains one of `V0V1`, `V1V0`, `V0V2`, `V2V0`, `V1V2`, `V2V1`, `V0F1`, `F1V0`, `V1F2`, `F2V1`, `V2F0`, or `F0V2`. Figure 4 illustrates four configurations that match the pseudocode (given after the figure).

Figure 4 Four configurations for the first time of contact. Although the labels use the original vertex names, the vertices are actually the original ones moved by their corresponding velocities.

(a) case V0V1

(b) case F0V2

(c) contact on left of U-triangle

(d) initially intersecting
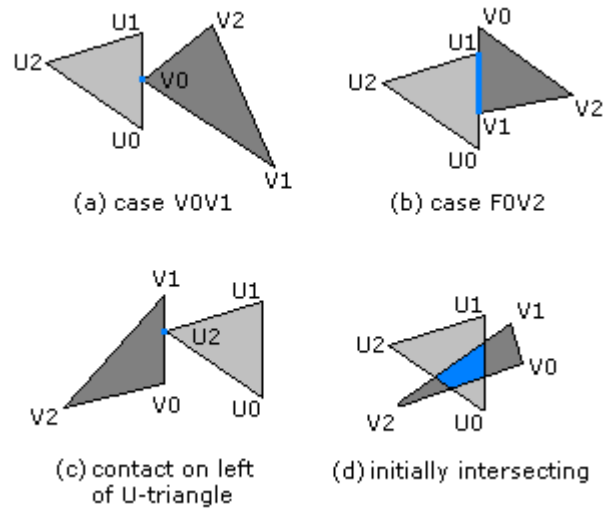
The following pseudocode shows how to construct the contact set. Again, the use of strings is just for illustration and most likely not how you would implement this.

```
if ( side == RIGHT )
{
    if ( TConfig [0] == 'V' )
    {
        // vertex−edge configuration    [ Figure 8.1 (a)]
        contactSet = { Vi + TFirst*W1 };    // i is number in TConfig[1]
    }
    else   // TConfig [0] == 'F'
    {
        // Edge−edge configuration .   See discussion after this code .
        // Vi0 is the first vertex and Vi1 is the last vertex of edge Fi
        // where i is the number in TConfig [1].   [ Figure 8.1 (b)]
        min = Dot(E0, Vi1−U0+TFirst*W)/Dot(E0,E0 );
        max = Dot(E0, Vi0−U0+TFirst*W)/Dot(E0,E0 );
        I = intersection ([0 ,1],[ min ,max ]);    // guaranteed not empty
        contactSet = U0 + TFirst*W1 + I*E0;   // point or line segment
    }
}
else if ( side == LEFT )
{
    // vertex−edge configuration    [ Figure 4 (c)]
```

12

```
        contactSet = { U2 + TFirst*W0 };
    }
    else  // triangles were initially intersecting
    {
        // Intersection set is a convex set: a point, a line segment, or a
        // convex polygon with at most six sides.  [Figure 8.1 (d)]
        Point UMove[3] = { U0+TFirst*W0, U1+TFirst*W0, U2+TFirst*W0 };
        Point VMove[3] = { V0+TFirst*W1, V1+TFirst*W1, V2+TFirst*W1 };
        contactSet = TriTriIntersection(UMove,VMove);
    }
```

In the case of edge-edge contact, after motion the two triangles touch at either a point or line segment. Let T denote the contact time. For the sake of argument, suppose that the contact edge for the second triangle is $\mathbf{F}_0$. The touching edges are parameterized by $\mathbf{U}_0 + T\mathbf{W}_1 + s\mathbf{E}_0$ for $s \in [0, 1]$ and $\mathbf{V}_0 + T\mathbf{W}_0 + s\mathbf{E}_0$ for $s \in [\mu_0, \mu_1]$ where

$$\mu_0 = \frac{(\mathbf{V}_1 + T\mathbf{W}_1) - (\mathbf{U}_0 + T\mathbf{W}_0)}{|\mathbf{E}_0|^2} \ \text{ and } \ \mu_1 = \frac{(\mathbf{V}_0 + T\mathbf{W}_1) - (\mathbf{U}_0 + T\mathbf{W}_0)}{|\mathbf{E}_0|^2}.$$

The overlap of the two edges occurs for $\bar{s} \in I = [0, 1] \cap [\mu_0, \mu_1] \neq \emptyset$. The corresponding points in the contact set are $\mathbf{U}_0 + T\mathbf{W}_0 + \bar{s}\mathbf{E}_0$.

In the event the two triangles are initially overlapping, the contact set is more expensive to construct. It can be a single point, a line segment, or a convex polygon with at most six sides. This set can be constructed by standard methods involving Boolean operations on polygons.

9 Example: Two Moving Triangles in 3D

Consider two triangles, $\langle \mathbf{U}_0, \mathbf{U}_1, \mathbf{U}_2 \rangle$ and $\langle \mathbf{V}_0, \mathbf{V}_1, \mathbf{V}_2 \rangle$. For the sake of indexing notation, define $\mathbf{U}_3 = \mathbf{U}_0$ and $\mathbf{V}_3 = \mathbf{V}_0$. The edges are $\mathbf{E}_i = \mathbf{U}_{i+1} - \mathbf{U}_i$ and $\mathbf{F}_i = \mathbf{V}_{i+1} - \mathbf{V}_i$ for $0 \leq i \leq 2$. A normal for the first triangle is $\mathbf{N} = \mathbf{E}_0 \times \mathbf{E}_1$ and a normal for the second triangle is $\mathbf{M} = \mathbf{F}_0 \times \mathbf{F}_1$. If the triangles are not coplanar, then the potential separating directions are $\mathbf{N}$, $\mathbf{M}$, and $\mathbf{E}_i \times \mathbf{F}_j$ for $0 \leq i \leq 2$ and $0 \leq j \leq 2$. If the triangles are parallel, but are not in the same plane, then $\mathbf{N}$ is a separating direction and the other directions need not be tested. Moreover, if $\mathbf{N}$ and $\mathbf{M}$ do not separate non-coplanar triangles, then the vectors $\mathbf{E}_i \times \mathbf{F}_j$ cannot be zero. If the triangles are coplanar, then the potential separating directions are $\mathbf{N} \times \mathbf{E}_i$ and $\mathbf{N} \times \mathbf{F}_i$ for $0 \leq i \leq 2$. This is exactly the 2D situation discussed earlier.

If $\mathbf{D}$ is a potential separating direction, then the block for computing the intervals of projection is more complex than that of its 2D counterpart. Both triangles are projected onto the separating axis. Each projection interval must be sorted to determine the appropriate configuration. The left/right relationship of the two projection intervals must be determined. The set of configurations for the projection of a single triangle consists of the following:

- **3**: All three vertices project to the same point. This happens when $\mathbf{D}$ is a normal vector to one of the triangles.

- **2-1**: Two vertices project to the minimum point of the interval, one vertex projects to the maximum point.

- **1-2**: One vertex projects to the minimum point of the interval, two vertices project to the maximum point.

13

- **1-1-1**: The three vertices project to distinct points in the interval.

The tables below summarize the possibilities. The symbols U, E, and N refer to the first triangle's vertices, edges, and full triangle. The indices after the U or E symbols indicate the specific vertex or edge that is involved in the contact. This information is calculated during the sorting of the projected points. The symbols V, F, and M refer to the second triangle's vertices, edges, and full triangle.

U left \ V right	3	2-1	1-2	1-1-1	V left \ U right	3	2-1	1-2	1-1-1
3	NM	NF_j	NV_j	NV_j	3	MN	ME_i	MU_i	MU_i
1-2	E_iM	E_iF_j	E_iV_j	E_iV_j	1-2	F_jN	F_jE_i	F_jU_i	F_jU_i
2-1	U_iM	U_iF_j	U_iV_j	U_iV_j	2-1	V_jN	V_jE_i	V_jU_i	V_jU_i
1-1-1	U_iM	U_iF_j	U_iV_j	U_iV_j	1-1-1	V_jN	V_jE_i	V_jU_i	V_jU_i

The intersection set for any table entry containing U_i or V_j is just that point. Each table contains 12 such cases. The intersection set for any table entry containing E_i or F_j but not containing U_i or V_j is a line segment (possibly degenerating to a point). Each table contains 3 such cases. Finally, the table entries NM and MN correspond to coplanar triangles that intersect. The intersection set is a point, line segment, or a convex polygon with at most six sides.

Pseudocode for handling the sorting and storing information for later code that determines contact sets is given below.

```
typedef enum { m3, m21, m12, m111 } ProjectionMap;

typedef struct
{
    ProjectionMap map;   // how vertices map to projection interval
    int index[3];        // the sorted indices of the vertices
    float min, max;      // the interval is [min,max]
}
Config;

Config GetConfiguration (Point D, Point U[3])
{
    // D is potential separating direction
    // U[3] are the triangle vertices

    Configuration config;
    d0 = Dot(D,U[0]), d1 = Dot(D,U[1]), d2 = Dot(D,U[2]);
    if ( d0 <= d1 )
    {
        if ( d1 <= d2 )   // d0 <= d1 <= d2
        {
            config.index[0] = 0;  config.index[1] = 1;  config.index[2] = 2;
            config.min = d0;   config.max = d2;
            if ( d0 != d1 )
                config.map = ( d1 != d2 ? m111 : m12 );
            else
                config.map = ( d1 != d2 ? m21 : m3 );
        }
        else if ( d0 <= d2 )   // d0 <= d2 < d1
        {
            config.index[0] = 0;  config.index[1] = 2;  config.index[2] = 1;
            config.min = d0;   config.max = d1;
            config.map = ( d0 != d2 ? m111 : m21 );
        }
        else   // d2 < d0 <= d1
        {
```

```
                config.index[0] = 2;   config.index[1] = 0;   config.index[2] = 1;
                config.min = d2;   config.max = d1;
                config.map = ( d0 != d1 ? m12 : m111 );
            }
        }
        else
        {
            if ( d2 <= d1 )   // d2 <= d1 < d0
            {
                config.index[0] = 2;   config.index[1] = 1;   config.index[2] = 0;
                config.min = d2;   config.max = d0;
                config.map = ( d2 != d1 ? m111 : m21 );
            }
            else if ( d2 <= d0 )   // d1 < d2 <= d0
            {
                config.index[0] = 1;   config.index[1] = 2;   config.index[2] = 0;
                config.min = d1;   config.max = d0;
                config.map = ( d2 != d0 ? m111 : m12 );
            }
            else   // d1 < d0 < d2
            {
                config.index[0] = 1;   config.index[1] = 0;   config.index[2] = 2;
                config.min = d1;   config.max = d2;
                config.map = m111;
            }
        }
        return config;
    }
```

Pseudocode for determining how the projection intervals relate to each other is given below.

```
    bool Update (Config UC, Config VC, float speed,
        Side& side, Config& TUC, Config& TVC, float& TFirst, float& TLast)
    {
        if ( VC.max < UC.min )   // V-interval initially on 'left' of U-interval
        {
            if ( speed <= 0 ) return false;   // intervals moving apart
            T = (UC.min - VC.max)/speed;
            if ( T > TFirst ) { TFirst = T; side = LEFT; TUC = UC; TVC = VC; }
            T = (UC.max - VC.min)/speed;   if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;
        }
        else if ( UC.max < VC.min )   // V-interval initially on 'right' of U-interval
        {
            if ( speed >= 0 ) return false;   // intervals moving apart
            T = (UC.max - VC.min)/speed;
            if ( T > TFirst ) { TFirst = T; side = RIGHT; TUC = UC; TVC = VC; }
            T=  (UC.min - VC.max)/speed;   if ( T < TLast ) TLast = T;
            if ( TFirst > TLast ) return false;
        }
        else   // U-interval and V-interval overlap
        {
            if ( speed > 0 )
            {
                T = (UC.max - VC.min)/speed;
                if ( T < TLast ) TLast = T;   if ( TFirst > TLast ) return false;
            }
            else if ( speed < 0 )
            {
                T = (UC.min - VC.max)/speed;
                if ( T < TLast ) TLast = T;   if ( TFirst > TLast ) return false;
            }
        }
        return true;
    }
```

It is assumed that the following routines exist for use in contact determination:

- Intersection of two line segments, call it SegSegIntersection.

- Intersection of line segment and triangle that are coplanar, call it `SegTriIntersection`.

- Intersection of triangle and triangle that are coplanar, call it `CoplanarTriTriIntersection`.

- Intersection of two stationary triangles, call it `TriTriIntersection`. This routine will contain a call to the coplanar intersection routine if the triangles happen to be coplanar.

Pseudocode for computing the contact set is given below.

```
ContactSet GetFirstContact (Point U[3], Point W0, Point V[3], Point W1,
    Side side, Config TUC, Config TVC, float TFirst)
{
    // move triangles to first contact
    Point UTri[3] = { U[0]+TFirst*W0, U[1]+TFirst*W0, U[2]+TFirst*W0 };
    Point VTri[3] = { V[0]+TFirst*W1, V[1]+TFirst*W1, V[2]+TFirst*W1 };
    Segment USeg, VSeg;

    if ( side == RIGHT )   // V-interval on right of U-interval
    {
        if ( TUC.map == m21 || TUC.map == m111 )
            return UTri[TUC.index[2]];

        if ( TVC.map == m12 || TVC.map == m111 )
            return VTri[TVC.index[0]];

        if ( TUC.map == m12 )
        {
            USeg = <UTri[TUC.index[1]], UTri[TUC.index[2]]>;
            if ( TVC.map == m21 )
            {
                VSeg = <VTri[TVC.index[0]], VTri[TVC.index[1]]>;
                return SegSegIntersection(USeg,VSeg);
            }
            else   // TVC.map == m3
            {
                return SegTriIntersection(USeg,VTri);
            }
        }
        else   // TUC.map == m3
        {
            if ( TVC.map == m21 )
            {
                VSeg = <VTri[TVC.index[0]], VTri[TVC.index[1]]>;
                return SegTriIntersection(VSeg,UTri);
            }
            else   // TVC.map == m3
            {
                return CoplanarTriTriIntersection(UTri,VTri);
            }
        }
    }
    else if ( side == LEFT )   // V-interval on left of U-interval
    {
        if ( TVC.map == m21 || TVC.map == m111 )
            return VTri[TVC.index[2]];

        if ( TUC.map == m12 || TUC.map == m111 )
            return UTri[TUC.index[0]];

        if ( TVC.map == m12 )
        {
            VSeg = <VTri[TVC.index[1]], VTri[TVC.index[2]]>;
            if ( TUC.map == m21 )
            {
                USeg = <UTri[TUC.index[0]], UTri[TUC.index[1]]>;
                return SegSegIntersection(USeg,VSeg);
            }
            else   // TUC.map == m3
            {
```

```
                    return SegTriIntersection(VSeg,UTri);
                }
            }
            else    // TVC.map == m3
            {
                if ( TUC.map == m21 )
                {
                    USeg = <UTri[TUC.index[0]],UTri[TUC.index[1]]>;
                    return SegTriIntersection(USeg,VTri);
                }
                else    // TUC.map == m3
                {
                    return CoplanarTriTriIntersection(UTri,VTri);
                }
            }
        }
    }
    else   // triangles were initially intersecting
    {
        return TriTriIntersection(UTri,VTri);
    }
}
```

The pseudocode that puts all this together is

```
bool TrianglesIntersect (Point U[3], Point W0, Point V[3], Point W1,
    float& TFirst, float& TLast, ContactSet& contact)
{
    W = W1 - W0;
    S = set of all potential separating axes;
    TFirst = 0;   TLast = INFINITY;
    side = NONE;
    Config TUC, TVC;

    for each D in S do
    {
        speed = Dot(D,W);
        Config UC = GetConfiguration(D,U);
        Config VC = GetConfiguration(D,V);
        if ( !Update(UC,VC,speed,side,TUC,TVC,TFirst,TLast) )
            return false;
    }

    contact = GetFirstContact(U,W0,V,W1,side,TUC,TVC,TFirst);
    return true;
}
```